CSCE 2014 – Programming Project 5

Midpoint Due Date – 7/27/17 at 11:59 PM
Final Due Date – 8/4/17 at 11:59 PM

1. Problem Statement:

This project will be an extension of your last assignment. To your simple calculator you will add support for several additional functions, including **min**, **max**, **sin**, **cos**, and **tan**. You will also add support for **variables**, allowing them to be used in calculations alongside numbers. Finally, you will add an interactive mode that allows users to enter calculations until they choose to exit the program. These collective upgrades will serve to reinforce your knowledge of the data structures that have already been discussed in the lecture, including **Stacks**, **Vectors**, and **Maps**.

For this assignment, you may either start with **your solution** from HW4 if you feel that it is complete enough, or you may start with the **posted solution** available on Moodle. You will be graded based on your implementations of the **new** features, not based on the specifics from the previous assignment.

1.1 Functions

Most calculators provide support for at least a few common functions. In this project, you will add support for **5**: **min**, **max**, **sin**, **cos**, and **tan**. For example, the user may enter the expressions like these:

```
>> max 2 3
3

>> min ( 2 3 )
2

>> sin 3.14159
2.65359e-006

>> cos ( 3 + ( 5 / 2 ) )
0.70867
```

Notice that neither **parentheses** nor **commas** are typically required when using functions. Parentheses may always be added if desired, though. One exception is for the **min**/**max** functions. If one or both of the arguments to the function are themselves expressions, the expressions should be encased in parentheses to ensure correct parsing. For example:

```
>> max ( 3 / 4 ) 5
5

>> max 3 / 4 5
Malformed expression
// (interpreted as "3 max 4/5" in postfix notation, which is meaningless)
```

It is important to differentiate between **unary** functions and **binary** ones. Unary functions require only a single argument, while binary functions require two. (There exist functions that take more than two

arguments, but we will not consider them for this assignment.) In this assignment, each of the **trigonometric** functions is unary and the **min** and **max** functions are binary.

You will need to modify both the **shunting yard** and **postfix evaluation** routines to add support for functions. The changes to the shunting yard algorithm are minimal. We simply must decide on which action to perform when we see a function token. We will treat function tokens in the same manner as left parentheses. They will simply be pushed onto the operator stack when seen. You must also ensure that functions have higher priority than the existing operators. The rest of the logic should remain unchanged.

The postfix evaluation routine will also require only a few small changes. You will now have to differentiate between **binary** operations and **unary** ones. For all binary operations, including the arithmetic operators used in the last assignment, as well as min and max, we must remove two items from the stack, perform the operation, and then push the result back onto the stack. For the unary operations, *only a single element is removed from the stack*, as unary operations only have one argument.

1.2 Variables

It is often helpful to be able to save the values of complex calculations in **variables** to simplify expressions. We would like to be able to enter assignment statements like these:

```
>> x = 5 + 3
x = 8

>> y = x - 2
y = 6

>> x
8

>> y
6

>> x = 7
x = 7
```

Each unique variable stores a single value. That value may be set and updated using **assignment statements**, which are characterized by a **variable name**, followed by an '**=**', followed by a **normal expression**. The expression on the right side is evaluated completely, which yields a single number. That number is then stored along with the variable name in a special **variable map**. The variable map is used to look up the value of the variable, given its name, in subsequent expressions. The name may be used in subsequent expressions as if it were any other number.

As in **Section 1.1**, several small changes will have to be made to both the **shunting yard** and **postfix expression evaluation** routines. For the shunting yard function, we will have to decide what to do with tokens that represent variables. Since the variables are simply aliases for concrete values, logically, we should apply the same process to them as we did with numbers. They will pass directly from the input to the output without interacting with the operator stack at all.

It is during the expression evaluation routine that the variable *name* will be converted to its corresponding *value*. If the current token represents a **variable name**, its value will be looked up in the **variable map**. That value will then be pushed onto the stack, just like any other number in the expression.

1.3 Interactivity

To make the program more interactive, you will need to allow the user to enter as many expressions or assignment statements as desired until he decides to exit the program. The following is a sample output:

```
Please enter either an individual expression, such as:
    sin ( 4 * 2 + 1 )
or a variable assignment, such as:
    x = (3 + 5) / 2

Variables that have previously been set may be used in
subsequent expression evaluations or assignments.

Enter Q to exit.

>> 3 + 5
8

>> x = ( 3 + 5 )
x = 8

>> y = sin ( 42 )
y = -0.916522

>> x + y
7.08348

>> Q
```

If a simple expression is entered, the program should simply print the result of the evaluation. If an assignment is entered, the program should print the variable name and the resulting value. If the user enters 'Q' (or something similar), the program should exit.

If the user enters an **invalid** expression, either because of mismatched parentheses or because the expression itself is malformed, the user's input should be **ignored** and they should be allowed to proceed. For example:

```
>> ( 3 + 5
Mismatched parentheses.

>> ( 3 + 5 )
8

>> Q
```

2. Design:

Generally, the design will be very similar to the previous assignment. You may use whichever means to solve the problem you desire, as long as your program meets the requirements listed above. You **are** allowed to use the data structures **designed in class** or their equivalents from the **C++ standard library**, and you may invent as many additional functions or classes as you like if it helps to improve the design. Lastly, be careful to avoid redundant or excessively long code if possible. Strive to write short, concise implementations.

3. Implementation:

As mentioned in **Section 1**, several changes will have to be made to the implementation of your previous program. The interfaces of both the **shuntingYard()** and **evaluatePostfix()** functions will have to be modified as well:

```
bool shuntingYard(const Vector<string>& expression,
    const int startIndex, Vector<string>& postfix)

bool evaluatePostfix(const Vector<string>& postfix,
    const Map<string, double>& variables, double& result)
```

The changes have been written in bold font. For **shuntingYard()**, you will need to pass the **starting index** to the function in addition to the original infix expression and the output postfix expression. This allows us to use the same code to evaluate both **simple expressions** and more the more complicated **assignment statements**. For simple expressions, we want to convert the entire expression, so we should start at index **0**. For assignment statements, we want to ignore the **first** and **second** tokens, since they represent the resulting variable name and the "**=**" token, which is useless in terms of evaluation.

For **evaluatePostfix()**, we need to include the variable map as an input. Values will be **inserted** into the variable map in the **driver**, and they will be retrieved in **evaluatePostfix()**. At any point in time, all variables that have been assigned by the program so far will be stored in the variable map.

3.1 Driver

You will also have to modify your **driver** quite a bit from the previous assignment. Apart from adding support for interactivity, you should also update the method used for reading and tokenizing the user's input. In the last assignment, we used a very simple approach to address this issue. The code looked like this:

```
Vector<string> expression;
string token;
while (cin >> token)
{
    expression.pushBack(token);
}
```

In this version, we exploited the the **>>** operator to split the input into tokens **directly**. However, this meant that the user was required to manually enter a breaking character (either **Ctrl + D** or **Ctrl + Z**) to stop the loop so evaluation could continue.

Doing so is a bit cumbersome for an interactive application, so for this assignment, we will improve the method slightly with the following:

```
#include <sstream>
// …

Vector<string> expression;
string str;
getline(cin, str);

stringstream ss(str);
while (ss >> str)
    expression.pushBack(str);
```

This version reads an **entire line** into a string, and then tokenizes it separately. The benefit to this approach is that the user need only enter their expression and press the enter key, which is a much simpler interface for them.

This code works by using the **getline()** function to read the line from cin and store the result in 'str'. Then, we use a special C++ object called a "**string stream**" to perform the tokenizing process. A string stream allows us to again make use of the **>>** operator to skip over whitespace, but it uses a normal string as its source instead of cin. The >> operator will fail when all tokens have been read from the stream, which stops the loop.

NOTE: Be careful to include the **<sstream>** library. Without it, you will not be able to use stringstreams in your code.

4. Style

Make sure your code adheres to the guidelines provided in the Style Guide (available on Moodle). Your goal is to create code that is concise, descriptive, and easy for other humans to read. Avoid typos, spelling mistakes, or anything else that degrades the aesthetic of your code. Your final submission should be work that you are proud to call your own.

5. Testing:

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values, and save your testing output in text files for inclusion in your project report.

6. Documentation:

When you have completed your C++ program, write a short report using the project report template describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report to be submitted electronically.

7. Project Submission:

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic plagiarism analysis of all programs that are submitted.

When you have completed the tasks above go to Moodle to upload your documentation (a single **.pdf** file), and all C++ program files (**.h** and **.cpp**). Make sure your proof of testing is included in the documentation or is submitted as a separate file. Do NOT upload an executable version of your program.

The dates on your electronic submission will be used to verify that you met the due date above. Late projects will receive **NO** credit. You will receive partial credit for all programs that compile even if they do not meet all program requirements, so make sure to submit something before the due date, even if the project is incomplete.

8. Academic Honesty Statement:

Students are expected to submit their own work on all programming projects, unless group projects have been explicitly assigned. Students are NOT allowed to distribute code to each other, or copy code from another individual or website. Students ARE allowed to use any materials on the class website, or in the textbook, or ask the instructor for assistance.

This course will be using highly effective program comparison software to calculate the similarity of all programs to each other, and to homework assignments from previous semesters. Please do not be tempted to plagiarize from another student.

Violations of the policies above will be reported to the Provost's office and may result in a **ZERO** on the programming project, an **F** in the class, or suspension from the university, depending on the severity of the violation and any history of prior violations.